Chapter 12
Data Structure

Associate Prof. Yuh-Shyan Chen
Dept. of Computer Science and Information Engineering
National Chung-Cheng University

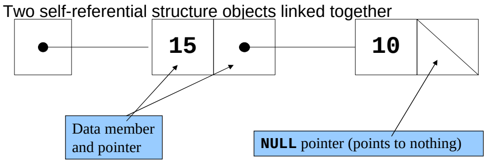
Outline

12.1 Introduction

- *Dynamic data structures* - grow and shrink during execution
  - Fixed-size data structure (single-subscripted array, double-subscripted array)
- *Linked lists* (Linear) - insertions and removals made anywhere
- *Stacks* (Linear) - insertions and removals made only at top of stack
- *Queues* (Linear) - insertions made at the back and removals made from the front
- *Binary trees* (Non-linear) - high-speed searching and sorting of data and efficient elimination of duplicate data items

12.2 Self-Referential Structures

- Self-referential structures
  - Structure that contains a pointer to a structure of the same type
  - Can be linked together to form useful data structures such as lists, queues, stacks and trees
  - Terminated with a `NULL` pointer (`0`)
- Two self-referential structure objects linked together



12.2 Self-Referential Classes (II)

```
struct node {
   int data;
   struct node *nextPtr;
}
```

- **`nextPtr`** - points to an object of type **node**
  - Referred to as a *link* – ties one **node** to another **node**

12.3 Dynamic Memory Allocation

- Dynamic memory allocation
  - Obtain and release memory during execution
- **`malloc`**
  - Takes number of bytes to allocate
    - Use **`sizeof`** to determine the size of an object
  - Returns pointer of type **`void *`**
    - type **`void *`** (*pointer to void*) to the allocated memory.
    - If no memory available, returns **`NULL`**
  - `newPtr = malloc( sizeof( struct node ) );`
- **`free`**
  - Deallocates memory allocated by **`malloc`**
  - Takes a pointer as an argument
  - `free (newPtr);`

12.4 Linked Lists

- Linked list
  - Linear collection of self-referential class objects, called *nodes*, connected by pointer *links*
  - Accessed via a pointer to the first node of the list
  - Subsequent nodes are accessed via the link-pointer member
  - Link pointer in the last node is set to null to mark the list's end
- Use a linked list instead of an array when
  - Number of data elements is unpredictable

An example of linked list

```
struct node {
   int data;
   struct node *nextPtr;
}
```

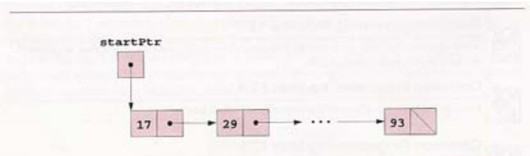


Fig. 12.2 A graphical representation of a linked list.

12.4 Linked Lists (II)

- Types of linked lists:
  - *singly linked list*
    - Begins with a pointer to the first node
    - Terminates with a null pointer
    - Only traversed in one direction
  - *circular, singly linked*
    - Pointer in the last node points back to the first node
  - *doubly linked list*
    - Two “start pointers”- first element and last element
    - Each node has a forward pointer and a backward pointer
    - Allows traversals both forwards and backwards
  - *circular, doubly linked list*
    - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

Inserting and deleting node in a list

```c
/* Fig. 12.3: fig12_03.c
   Operating and maintaining a list */
#include <stdio.h>
#include <stdlib.h>

struct listNode {    /* self-referential structure */
   char data;
   struct listNode *nextPtr;
};

typedef struct listNode ListNode;
typedef ListNode *ListNodePtr;

void insert( ListNodePtr *, char );
char delete( ListNodePtr *, char );
int isEmpty( ListNodePtr );
void printList( ListNodePtr );
void instructions( void );

```

Main program

```c

int main()
{
   ListNodePtr startPtr = NULL;
   int choice;
   char item;

   instructions();  /* display the menu */
   printf( "? " );
   scanf( "%d", &choice );

   while ( choice != 3 ) {

      switch ( choice ) {
         case 1:
            printf( "Enter a character: " );
            scanf( "\n%c", &item );
            insert( &startPtr, item );
            printList( startPtr );
            break;
```

```
Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 1
Enter a character: B
The list is:
B --> NULL

? 1
Enter a character: A
The list is:
A --> B --> NULL

? 1
Enter a character: C
The list is:
A --> B --> C --> NULL

? 2
Enter character to be deleted: D
D not found.

? 2
Enter character to be deleted: B
B deleted.
The list is:
A --> C --> NULL
```

```c
        case 2:
            if ( !isEmpty( startPtr ) ) {
                printf( "Enter character to be deleted: " );
                scanf( "\n%c", &item );

                if ( delete( &startPtr, item ) ) {
                    printf( "%c deleted.\n", item );
                    printList( startPtr );
                }
                else
                    printf( "%c not found.\n\n", item );
            }
            else
                printf( "List is empty.\n\n" );

            break;
        default:
            printf( "Invalid choice.\n\n" );
            instructions();
            break;
      }

      printf( "? " );
      scanf( "%d", &choice );
   }

   printf( "End of run.\n" );
   return 0;
}
```

Function instructions

```c
/* Print the instructions */
void instructions( void )
{
   printf( "Enter your choice:\n"
      "   1 to insert an element into the list.\n"
      "   2 to delete an element from the list.\n"
      "   3 to end.\n" );
}
```

```c
/* Insert a new value into the list in sorted order */
void insert( ListNodePtr *sPtr, char value )
{
   ListNodePtr newPtr, previousPtr, currentPtr;

   newPtr = malloc( sizeof( ListNode ) );

   if ( newPtr != NULL ) {     /* is space available */
      newPtr->data = value;
      newPtr->nextPtr = NULL;

      previousPtr = NULL;
      currentPtr = *sPtr;

      while ( currentPtr != NULL && value > currentPtr->data ) {
         previousPtr = currentPtr;          /* walk to ...   */
         currentPtr = currentPtr->nextPtr;  /* ... next node */
      }

      if ( previousPtr == NULL ) {
         newPtr->nextPtr = *sPtr;
         *sPtr = newPtr;
      }
      else {
         previousPtr->nextPtr = newPtr;
         newPtr->nextPtr = currentPtr;
      }
   }
   else
      printf( "%c not inserted. No memory available.\n", value );
}
```

Inserting a node in order in a list

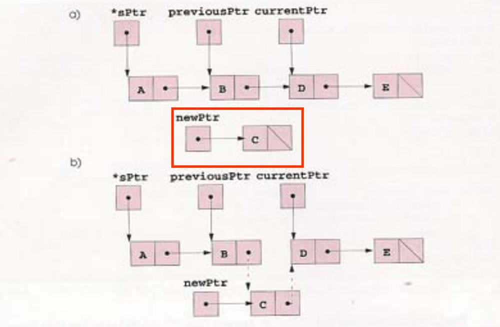


Fig. 12.5 Inserting a node in order in a list.

```c
/* Delete a list element */
char delete( ListNodePtr *sPtr, char value )
{
   ListNodePtr previousPtr, currentPtr, tempPtr;

   if ( value == ( *sPtr )->data ) {
      tempPtr = *sPtr;
      *sPtr = ( *sPtr )->nextPtr;  /* de-thread the node */
      free( tempPtr );             /* free the de-threaded node */
      return value;
   }
   else {
      previousPtr = *sPtr;
      currentPtr = ( *sPtr )->nextPtr;

      while ( currentPtr != NULL && currentPtr->data != value ) {
         previousPtr = currentPtr;          /* walk to ...   */
         currentPtr = currentPtr->nextPtr;  /* ... next node */
      }

      if ( currentPtr != NULL ) {
         tempPtr = currentPtr;
         previousPtr->nextPtr = currentPtr->nextPtr;
         free( tempPtr );
         return value;
      }
   }

   return '\0';
}
```

Deleting a node from a list

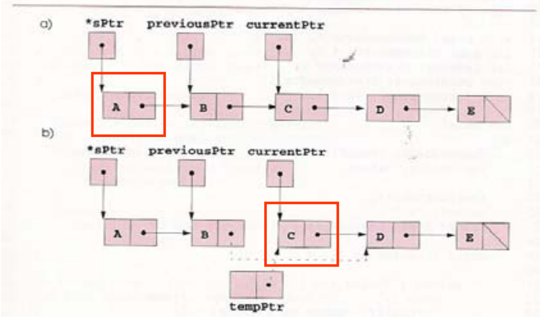


Fig. 12.6 Deleting a node from a list.

Print out a list

```c
}

/* Return 1 if the list is empty, 0 otherwise */
int isEmpty( ListNodePtr sPtr )
{
   return sPtr == NULL;
}

/* Print the list */
void printList( ListNodePtr currentPtr )
{
   if ( currentPtr == NULL )
      printf( "List is empty.\n\n" );
   else {
      printf( "The list is:\n" );

      while ( currentPtr != NULL ) {
         printf( "%c --> ", currentPtr->data );
         currentPtr = currentPtr->nextPtr;
      }

      printf( "NULL\n\n" );
   }
}
```

```
Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 1
Enter a character: B
The list is:
B --> NULL

? 1
Enter a character: A
The list is:
A --> B --> NULL

? 1
Enter a character: C
The list is:
A --> B --> C --> NULL

? 2
Enter character to be deleted: D
D not found.

? 2
Enter character to be deleted: B
B deleted.
The list is:
A --> C --> NULL
```

12.5 Stacks

- Stack
  - New nodes can be added and removed only at the top
  - Similar to a pile of dishes
  - Last-in, first-out (LIFO)
  - Bottom of stack indicated by a link member to NULL
  - Constrained version of a linked list
- push
  - Adds a new node to the top of the stack
- pop
  - Removes a node from the top
  - Stores the popped value
  - Returns **`true`** if **`pop`** was successful

Graphical representation of a stack

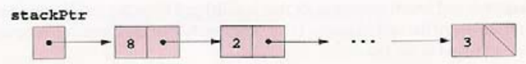


Fig. 12.7 Graphical representation of a stack.

A simple stack program

```c
/* Fig. 12.8: fig12_08.c
   dynamic stack program */
#include <stdio.h>
#include <stdlib.h>

struct stackNode {   /* self-referential structure */
   int data;
   struct stackNode *nextPtr;
};

typedef struct stackNode StackNode;
typedef StackNode *StackNodePtr;

void push( StackNodePtr *, int );
int pop( StackNodePtr * );
int isEmpty( StackNodePtr );
void printStack( StackNodePtr );
void instructions( void );
```

```c
int main()
{
   StackNodePtr stackPtr = NULL;  /* points to stack top */
   int choice, value;

   instructions();
   printf( "? " );
   scanf( "%d", &choice );

   while ( choice != 3 ) {

      switch ( choice ) {
         case 1:       /* push value onto stack */
            printf( "Enter an integer: " );
            scanf( "%d", &value );
            push( &stackPtr, value );
            printStack( stackPtr );
            break;
         case 2:       /* pop value off stack */
            if ( !isEmpty( stackPtr ) )
               printf( "The popped value is %d.\n",
                       pop( &stackPtr ) );

            printStack( stackPtr );
            break;
         default:
            printf( "Invalid choice.\n\n" );
            instructions();
            break;
      }

      printf( "? " );
      scanf( "%d", &choice );
   }

   printf( "End of run.\n" );
   return 0;
}

/* Print the instructions */
void instructions( void )
{
   printf( "Enter choice:\n"
           "1 to push a value on the stack\n"
           "2 to pop a value off the stack\n"
           "3 to end program\n" );
}
```

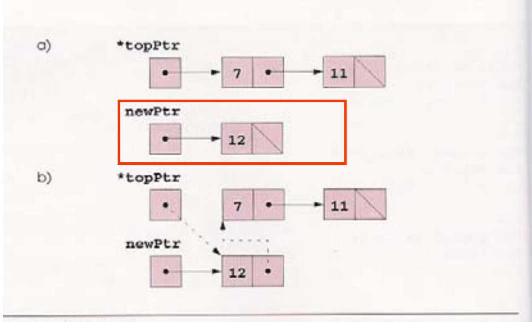
```
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 1
Enter an integer: 5
The stack is:
5 --> NULL

? 1
Enter an integer: 6
The stack is:
6 --> 5 --> NULL

? 1
Enter an integer: 4
The stack is:
4 --> 6 --> 5 --> NULL

? 2
The popped value is 4.
The stack is:
6 --> 5 --> NULL
```

```
? 2
The popped value is 6.
The stack is:
5 --> NULL

? 2
The popped value is 5.
The stack is empty.

? 2
The stack is empty.

? 4
Invalid choice.

Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 3
End of run.
```

Function push

```c
/* Insert a node at the stack top */
void push( StackNodePtr *topPtr, int info )
{
   StackNodePtr newPtr;

   newPtr = malloc( sizeof( StackNode ) );
   if ( newPtr != NULL ) {
      newPtr->data = info;
      newPtr->nextPtr = *topPtr;
      *topPtr = newPtr;
   }
   else
      printf( "%d not inserted. No memory available.\n",
              info );
}
```

Push operation

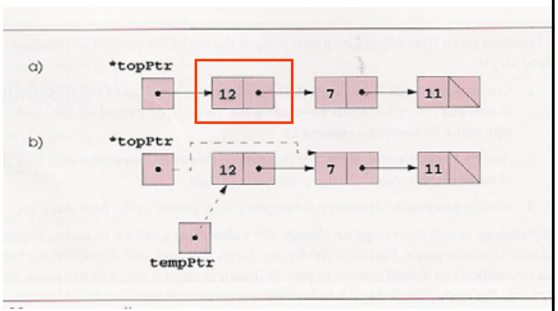


ush operation.

Function pop

```c
/* Remove a node from the stack top */
int pop( StackNodePtr *topPtr )
{
    StackNodePtr tempPtr;
    int popValue;

    tempPtr = *topPtr;
    popValue = ( *topPtr )->data;
    *topPtr = ( *topPtr )->nextPtr;
    free( tempPtr );
    return popValue;
}
```

Pop operation




```c
/* Print the stack */
void printStack( StackNodePtr currentPtr )
{
    if ( currentPtr == NULL )
        printf( "The stack is empty.\n\n" );
    else {
        printf( "The stack is:\n" );

        while ( currentPtr != NULL ) {
            printf( "%d --> ", currentPtr->data );
            currentPtr = currentPtr->nextPtr;
        }

        printf( "NULL\n\n" );
    }
}

/* Is the stack empty? */
int isEmpty( StackNodePtr topPtr )
{
    return topPtr == NULL;
}
```

Applications of Stack

- Stacks support recursive function calls

12.6 Queues

- Queue
  - Similar to a supermarket checkout line
  - *First-in, first-out (FIFO)*
  - Nodes are removed only from the *head*
  - Nodes are inserted only at the *tail*
- Insert and remove operations
  - Enqueue (insert) and dequeue (remove)
- Useful in computing
  - Print spooling, packets in networks, file server requests

A graphical representation of a queue

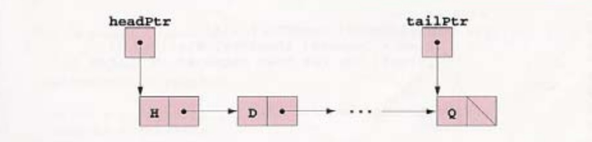


Fig. 12.12 A graphical representation of a queue.

Example of queue operation

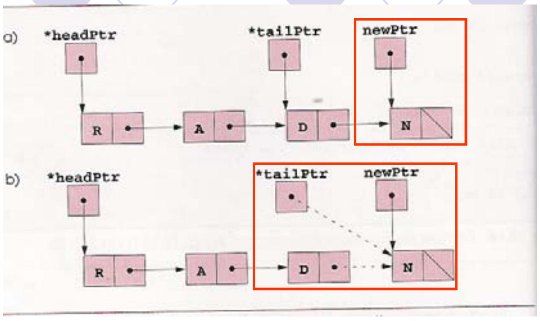
```c
/* Fig. 12.13: fig12_13.c
   Operating and maintaining a queue */

#include <stdio.h>
#include <stdlib.h>

struct queueNode {   /* self-referential structure */
   char data;
   struct queueNode *nextPtr;
};

typedef struct queueNode QueueNode;
typedef QueueNode *QueueNodePtr;

/* function prototypes */
void printQueue( QueueNodePtr );
int isEmpty( QueueNodePtr );
char dequeue( QueueNodePtr *, QueueNodePtr * );
void enqueue( QueueNodePtr *, QueueNodePtr *, char );
void instructions( void );
```

```c
int main()
{
   QueueNodePtr headPtr = NULL, tailPtr = NULL;
   int choice;
   char item;

   instructions();
   printf( "? " );
   scanf( "%d", &choice );

   while ( choice != 3 ) {

      switch( choice ) {

         case 1:
            printf( "Enter a character: " );
            scanf( "\n%c", &item );
            enqueue( &headPtr, &tailPtr, item );
            printQueue( headPtr );
            break;
         case 2:
            if ( !isEmpty( headPtr ) ) {
               item = dequeue( &headPtr, &tailPtr );
               printf( "%c has been dequeued.\n", item );
            }

            printQueue( headPtr );
            break;

         default:
            printf( "Invalid choice.\n\n" );
            instructions();
            break;
      }

      printf( "? " );
      scanf( "%d", &choice );
   }

   printf( "End of run.\n" );
   return 0;
}

void instructions( void )
{
   printf ( "Enter your choice:\n"
           "   1 to add an item to the queue\n"
           "   2 to remove an item from the queue\n"
           "   3 to end\n" );
}
```

enqueue function

```c
void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
              char value )
{
   QueueNodePtr newPtr;

   newPtr = malloc( sizeof( QueueNode ) );

   if ( newPtr != NULL ) {
      newPtr->data = value;
      newPtr->nextPtr = NULL;

      if ( isEmpty( *headPtr ) )
         *headPtr = newPtr;
      else
         ( *tailPtr )->nextPtr = newPtr;

      *tailPtr = newPtr;
   }
   else
      printf( "%c not inserted. No memory available.\n",
              value );
}
```

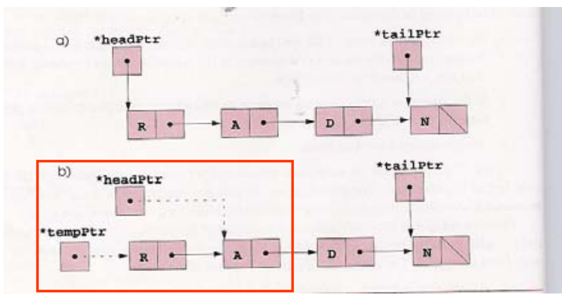
enqueue operation



dequeue function

```c
char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr )
{
   char value;
   QueueNodePtr tempPtr;

   value = ( *headPtr )->data;
   tempPtr = *headPtr;
   *headPtr = ( *headPtr )->nextPtr;

   if ( *headPtr == NULL )
      *tailPtr = NULL;

   free( tempPtr );
   return value;
}
```

dequeue operation



Other functions

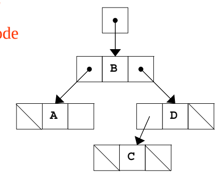
```c
int isEmpty( QueueNodePtr headPtr )
{
   return headPtr == NULL;
}

void printQueue( QueueNodePtr currentPtr )
{
   if ( currentPtr == NULL )
      printf( "Queue is empty.\n\n" );
   else {
      printf( "The queue is:\n" );

      while ( currentPtr != NULL ) {
         printf( "%c --> ", currentPtr->data );
         currentPtr = currentPtr->nextPtr;
      }

      printf( "NULL\n\n" );
   }
}
```

```
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 1
Enter a character: A
The queue is:
A --> NULL

? 1
Enter a character: B
The queue is:
A --> B --> NULL

? 1
Enter a character: C
The queue is:
A --> B --> C --> NULL
```

```
? 2
A has been dequeued.
The queue is:
B --> C --> NULL

? 2
B has been dequeued.
The queue is:
C --> NULL

? 2
C has been dequeued.
Queue is empty.

? 2
Queue is empty.

? 4
Invalid choice.

Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 3
End of run.
```

12.7 Trees

- Tree nodes contain two or more links
  - All other data structures we have discussed only contain one
- Binary trees
  - All nodes contain two links (connecting with left and right child nodes)
    - None, one, or both of which may be **NULL**
  - The *root node* is the first node in a tree.
  - Each link in the root node refers to a *child*
  - A node with no children is called a leaf node

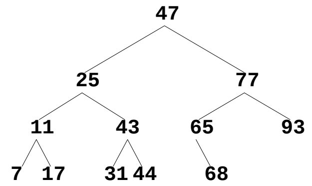


Tree node structure

```c
/* Fig. 12.19: fig12_19.c
   Create a binary tree and traverse it
   preorder, inorder, and postorder */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct treeNode {
   struct treeNode *leftPtr;
   int data;
   struct treeNode *rightPtr;
};

typedef struct treeNode TreeNode;
typedef TreeNode *TreeNodePtr;

void insertNode( TreeNodePtr *, int );
void inOrder( TreeNodePtr );
void preOrder( TreeNodePtr );
void postOrder( TreeNodePtr );
```

12.7 Trees (II)

- Binary search tree
  - Values in left subtree less than parent
  - Values in right subtree greater than parent
  - Facilitates *duplicate elimination*
  - Fast searches - for a balanced tree, maximum of $\log_2 n$ comparisons



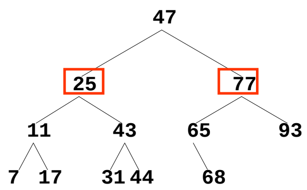
main function

```c
int main()
{
   int i, item;
   TreeNodePtr rootPtr = NULL;

   srand( time( NULL ) );

   /* insert random values between 1 and 15 in the tree */
   printf( "The numbers being placed in the tree are:\n" );

   for ( i = 1; i <= 10; i++ ) {
      item = rand() % 15;
      printf( "%3d", item );
      insertNode( &rootPtr, item );
   }

   /* traverse the tree preOrder */
   printf( "\n\nThe preOrder traversal is:\n" );
   preOrder( rootPtr );

   /* traverse the tree inOrder */
   printf( "\n\nThe inOrder traversal is:\n" );
   inOrder( rootPtr );

   /* traverse the tree postOrder */
   printf( "\n\nThe postOrder traversal is:\n" );
   postOrder( rootPtr );

   return 0;
}
```

insertNode function

```c
void insertNode( TreeNodePtr *treePtr, int value )
{
   if ( *treePtr == NULL ) {   /* *treePtr is NULL */
      *treePtr = malloc( sizeof( TreeNode ) );

      if ( *treePtr != NULL ) {
         ( *treePtr )->data = value;
         ( *treePtr )->leftPtr = NULL;
         ( *treePtr )->rightPtr = NULL;
      }
      else
         printf( "%d not inserted. No memory available.\n",
                 value );
   }
   else
      if ( value < ( *treePtr )->data )
         insertNode( &( ( *treePtr )->leftPtr ), value );
      else if ( value > ( *treePtr )->data )
         insertNode( &( ( *treePtr )->rightPtr ), value );
      else
         printf( "dup" );
}
```

Example of a binary search tree



12.7 Trees (III)

- Tree traversals:
  - Inorder traversal - prints the node values in ascending order
    1. Traverse the left subtree with an inorder traversal.
    2. Process the value in the node (i.e., print the node value).
    3. Traverse the right subtree with an inorder traversal.
  - Preorder traversal:
    1. Process the value in the node.
    2. Traverse the left subtree with a preorder traversal.
    3. Traverse the right subtree with a preorder traversal.
  - Postorder traversal:
    1. Traverse the left subtree with a postorder traversal.
    2. Traverse the right subtree with a postorder traversal.
    3. Process the value in the node.

```c
void inOrder( TreeNodePtr treePtr )
{
   if ( treePtr != NULL ) {
      inOrder( treePtr->leftPtr );
      printf( "%3d", treePtr->data );
      inOrder( treePtr->rightPtr );
   }
}

void preOrder( TreeNodePtr treePtr )
{
   if ( treePtr != NULL ) {
      printf( "%3d", treePtr->data );
      preOrder( treePtr->leftPtr );
      preOrder( treePtr->rightPtr );
   }
}

void postOrder( TreeNodePtr treePtr )
{
   if ( treePtr != NULL ) {
      postOrder( treePtr->leftPtr );
      postOrder( treePtr->rightPtr );
      printf( "%3d", treePtr->data );
   }
}
```

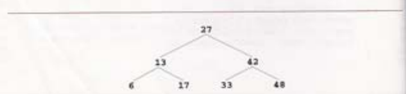


Fig. 12.21 A binary search tree.

```
The preOrder traversal is:
  27 13 6 17 42 33 48

The inOrder traversal is:
  6 13 17 27 33 42 48

The postOrder traversal is:
  6 17 13 33 48 42 27
```